Engineer

Databaseindeks

en struktur, der finder rækker uden at læse hele tabellen

Hvad det er

Et databaseindeks er en separat struktur, gemt ved siden af en tabel, som lader databasen gå direkte til de rækker, der opfylder en betingelse, i stedet for at læse tabellen fra første til sidste række. Til daglig, og i det meste af PostgreSQL-dokumentationen, hedder det bare et indeks. Dokumentationen sammenligner det med stikordsregistret bag i en bog: den, der leder efter ét emne, slår op dér og bladrer direkte til de sider, der står, i stedet for at læse hele bogen igennem.

Tag en tabel med ordrer og betingelsen `WHERE customer_id = 42`. Uden et indeks på `customer_id` har PostgreSQL kun én måde at svare på: en sekventiel scanning, der læser hver eneste række og beholder dem, der passer. Det koster det samme, uanset om kunden har to ordrer eller to tusind, og prisen vokser med tabellen. Med et indeks på kolonnen går PostgreSQL et par niveauer ned gennem et søgetræ til posterne for 42 og henter kun de rækker. Et indeks oprettes med én sætning, `DROP INDEX` fjerner det igen, og begge dele kan ske når som helst.

Når indekset først findes, holder PostgreSQL det ajour med tabellen ved hver skrivning, og forespørgselsplanlæggeren afgør fra forespørgsel til forespørgsel, om det er billigere at bruge det end at scanne. Intet i SQL'en nævner indekset, så det samme indeks kan gøre en `SELECT` hurtigere, ligesom en `UPDATE` eller `DELETE` med en søgebetingelse og et join.

PostgreSQL har flere slags, hver bygget til sin type spørgsmål:

- **B-tree** er standarden og det, `CREATE INDEX` bygger, medmindre man beder om andet. Det besvarer lighed og intervaller (`<` , `<=` , `=` , `>=` , `>`), `BETWEEN` , `IN` , `IS NULL` og en `LIKE` , hvis mønster er forankret i starten. Det holder sine poster sorteret og kan derfor også levere rækkerne i orden.
- **Hash** gemmer en 32-bit hashkode for hver værdi og besvarer kun lighed.

- **GiST** og **SP-GiST** er rammer for data som geometri, hvor spørgsmålet er »overlapper«, »indeholder« eller »nærmest ved« snarere end »er lig med«.
- **GIN** er et inverteret indeks til værdier, der består af mange dele, for eksempel arrays, og finder de rækker, hvis værdi indeholder et bestemt element.
- **BRIN** gemmer kun et resumé for hvert interval af tabelblokke. Det passer til en kolonne, hvis værdier følger rækkernes fysiske rækkefølge, sådan som et indsættelsestidspunkt ofte gør.

Ud over typen kan et indeks formes efter spørgsmålet. Et unikt indeks afviser dubletter. Et sammensat indeks dækker flere kolonner, op til 32. Et partielt indeks har en `WHERE`-klausul og rummer kun de rækker, der opfylder den. Et udtryksindeks gemmer en beregnet værdi som `lower(email)`. Og en `INCLUDE`-liste tager ekstra kolonner med i indekset uden at søge på dem, så en forespørgsel kan besvares fra indekset alene.

Brug begrebet i en beslutning

Brug »Hvad er et databaseindeks?« i en konkret beslutning. Vi kan knytte begrebet til det system, I bygger eller køber. At finde de forespørgsler og manglende indeks bag et system, der er blevet tungere hvert år, og rette dem, der betyder noget.

Se vores ydelser Kontakt os →

Undersøg antagelserne bag ordet

Brug forklaringen og det tilknyttede arbejde til at formulere spørgsmål til jeres projekt. Vi kan undersøge antagelserne sammen, før et værktøj vælges. 30 dages eldata på 180 ms — 180 ms til at aggregere en måneds måleraflæsninger

30 dages eldata på 180 ms Kontakt os →

Hvorfor det betyder noget

Forskellen, et indeks gør, vokser med tabellen. På en lille tabel er en scanning billig, og planlæggeren vælger ofte, med rette, netop den: at læse en håndfuld sider i rækkefølge koster mindre end at gå gennem et træ for at nå dem. Med et tocifret antal millioner rækker er forskellen mellem en scanning og et opslag forskellen mellem et svar med det samme og en timeout. En forespørgsel, der var hurtig ved lanceringen, kan blive langsom et år senere, uden at en eneste kodelinje er ændret, blot fordi tabellen, den læser, er vokset.

Indekser er ikke gratis, og regningen kommer ved skrivning. Hver insert tilføjer en post til hvert indeks på tabellen. En opdatering kan kun slippe for det som en heap-only tuple-opdatering (HOT), hvor den nye rækkeversion bliver på samme side og ikke kræver nye indeksposter, og det tillader PostgreSQL kun, når opdateringen ikke ændrer nogen kolonne, som tabellens indekser bruger. BRIN-

indekser tæller ikke med, fordi de kun gemmer et resumé. Hvert indeks gør altså den billige vej smallere, og derfor råder dokumentationen til at fjerne indekser, der sjældent eller aldrig bliver brugt. En tabel med et dusin indekser betaler et dusin gange ved hver insert.

Det koster også at bygge et indeks. En almindelig `CREATE INDEX` lader andre sessioner læse tabellen, men blokerer dens skrivninger, indtil opbygningen er færdig, og dokumentationen advarer om, at det på en meget stor tabel kan tage mange timer.

Endelig er et indeks kun så godt som planlæggerens beslutning om at bruge det, og planlæggeren arbejder ud fra statistik, som `ANALYZE` har samlet. Spørgsmålet er ikke »findes der et indeks?«, men »bruger de forespørgsler, der betyder noget, det?«, og `EXPLAIN` er måden at finde ud af det på.

Sådan ser det ud i praksis

En webshops kontoside viser en kundes tyve seneste ordrer:

```
SELECT id, created_at, total
FROM   orders
WHERE  customer_id = 42
ORDER  BY created_at DESC
LIMIT  20;
```

Med kun en primærnøgle på `id` må PostgreSQL læse hele `orders`, beholde kundens rækker, sortere dem og smide alle undtagen tyve væk. `EXPLAIN` viser det: en `Seq Scan` på `orders` med betingelsen som `Filter` og en sortering ovenover. `EXPLAIN ANALYZE` kører også forespørgslen og skriver de faktiske rækkeantal og tider ved siden af estimaterne.

Det indeks, der passer til spørgsmålet, har lighedskolonnen først og sorteringskolonnen bagefter:

```
CREATE INDEX CONCURRENTLY orders_customer_created_idx
ON orders (customer_id, created_at);
```

Kør `EXPLAIN` igen, og scanningen bliver en indeksscanning med `orders_customer_created_idx`, hvor `customer_id = 42` nu står som `Index Cond` i stedet for som filter. Sorteringen forsvinder også. Et B-tree holder posterne for kunde 42 ordnet efter `created_at`, PostgreSQL kan læse dem baglæns for at få de nyeste først, og med `LIMIT` stopper den efter tyve poster uden at røre resten.

Kolonnernes rækkefølge betyder noget. Det samme indeks dækker en forespørgsel på `customer_id` alene, fordi det er den forreste kolonne, men en rapport over alle gårsdagens ordrer, der kun filtrerer på `created_at`, får langt mindre ud af det. I et sammensat B-tree er det lighed på de forreste kolonner plus et interval på den næste, der indsnævrer den del af indekset, som bliver læst. PostgreSQL 18 kan lave skip scan og gentage søgningen én gang for hver forskellig værdi i den forreste kolonne, men planlæggeren vælger normalt kun det, når der er få forskellige værdier, og kunde-id'er er der mange af. Betyder rapporten noget, får den sit eget indeks på `created_at`. Vendt om, som `(created_at, customer_id)`, ville indekset sprede kunde 42's ordrer ud over det hele.

Noten om at optimere databasen, før du sharder den sætter indekser først blandt de forbedringer, den ordner efter udbytte, når planerne har udpeget de scanninger, en ydelseskritisk forespørgsel kører. Dens eget eksempel er et partielt indeks, `units_unsold`, der kun rummer de enheder, der stadig er til salg, så det skrumper, mens et salg skrider frem, i stedet for at vokse. Noten regner også prisen ud. På dens tabel var 2.000 reservationer, der satte `reserved`, `reserved_at` og `order_id`, alle HOT-opdateringer, mens ingen af 2.000 opdateringer, der satte `sold`, en kolonne det partielle indeks læser, var det. Dens råd er at fjerne de indekser, hvis `idx_scan` i `pg_stat_user_indexes` bliver ved med at stå på nul.

Casen om et bedre valg af nøgle viser, hvorfor de værdier, der havner i et B-tree, betyder lige så meget som kolonnerne. Tilfældige UUID'er spreder inserts ud over indekset, hvilket kan skabe fragmentering. Platformen gjorde tidsordnede UUID v7 til standard for entiteternes primærnøgler, med et tidsstempel i de forreste bits, så nye identifikatorer rummer en tidsmæssig rækkefølge. Mindre fragmentering og hurtigere forespørgsler var designmål; det dokumenterede resultat er den konfigurerede nøglekonvention, og casen dokumenterer ikke et benchmark, der måler ydelsen før og efter under en arbejdsbelastning.

Hvor det går galt

Et nyt indeks stopper tabellens skrivninger. En almindelig `CREATE INDEX` på en travl tabel blokerer inserts, opdateringer og sletninger, så længe opbygningen varer, og en rutinemæssig udrulning bliver til et nedbrud. Brug `CREATE INDEX CONCURRENTLY` : den scanner tabellen to gange og venter på transaktioner, der kunne bruge indekset, så den tager betydeligt længere tid, men skrivningerne fortsætter. Den kan ikke køre inde i en transaktionsblok, så et migreringsværktøj, der pakker hvert trin ind i én, skal have besked på at køre netop den sætning for sig. Der kan kun køre én samtidig opbygning på en tabel ad gangen.

En samtidig opbygning fejler og efterlader et ugyldigt indeks. Et deadlock eller en dubletværdi under opbygningen af et unikt indeks får `CREATE INDEX CONCURRENTLY` til at fejle, men det påbegyndte indeks bliver liggende i kataloget, markeret som ugyldigt. Forespørgsler ignorerer det, men hver skrivning betaler stadig for at vedligeholde det, og et ugyldigt unikt indeks kan blive ved med at håndhæve unikhed. psql's `\d` viser det med `INVALID` efter definitionen. Fjern det, og kør den samtidige opbygning igen, eller genopbyg det med `REINDEX INDEX CONCURRENTLY` .

Forespørgslen gemmer kolonnen inde i en funktion. Et indeks på `email` gør intet for `WHERE lower(email) = 'ana@example.com'` , fordi indekset rummer `email` , ikke `lower(email)` . Indeksér det udtryk, forespørgslen bruger:

```
CREATE INDEX users_lower_email_idx ON users (lower(email));
```

Udtrykket beregnes ved hver insert og hver opdatering, der ikke er HOT, og derfor anbefaler dokumentationen udtryksindekser dér, hvor hurtige opslag betyder mere end hurtige skrivninger.

Der hober sig indekser op, som ingen bruger. Hvert indeks, der er lagt på »for en sikkerheds skyld«, koster ved hver skrivning og kan spærre for HOT-opdateringer. `pg_stat_user_indexes` tæller de scanninger, der er startet på hvert indeks, i `idx_scan` og noterer tidspunktet for den seneste i `last_idx_scan` :

```
SELECT relname, indexrelname, idx_scan, last_idx_scan
FROM   pg_stat_user_indexes
WHERE  idx_scan = 0
```

```
ORDER  BY relname;
```

Se efter, før du sletter. Et indeks bag en primærnøgle eller en unik constraint gør sit arbejde, uanset om en forespørgsel nogensinde læser det, og et nul siger ikke meget på en server, der lige er begyndt at tage imod trafik.

Planlæggeren ignorerer et godt indeks. Efter en stor indlæsning halter statistikken efter dataene, estimaterne rammer forkert, og en scanning vinder på papiret. Kør `ANALYZE` ; dokumentationen kalder det håbløst at vurdere indeksbrug uden. Test også på realistiske mængder: på hundrede rækker, der kan ligge på én diskside, slår intet indeks det at læse den ene side. Vil du vide, om et indeks overhovedet kan bruges, så sæt `enable_seqscan = off` i en testsession, og kør `EXPLAIN` igen. Scanner planen stadig, passer betingelsen ikke til indekset.

Læs mere

- Indexes: Introduction — hvad et indeks er til, hvordan det oprettes og fjernes, og hvad det koster ved hver skrivning.
- Index Types — B-tree, Hash, GiST, SP-GiST, GIN og BRIN, og hvilke operatorer hver af dem kan besvare.
- Multicolumn Indexes — hvordan kolonnernes rækkefølge afgør, hvad et B-tree kan indsnævre, og hvornår skip scan kommer i spil.
- CREATE INDEX — hele syntaksen, herunder `CONCURRENTLY` , `INCLUDE` , partielle indekser og oprydning efter en ugyldig opbygning.
- Using EXPLAIN — at læse en plan: scanningstyper, estimerede omkostninger og de faktiske tal fra `EXPLAIN ANALYZE` .
- Examining Index Usage — hvorfor `ANALYZE` og rigtige data kommer først, og hvordan man tester, om et indeks kan bruges.

Ordliste

Vælg ud fra jeres faktiske vilkår

Medbring et repræsentativt datasæt, de spørgsmål det skal besvare, og den aktuelle flaskehals. Vi kan aftale målinger af kvalitet, gennemløb og gendannelse.

Se, hvad det koster Kontakt os →

Gør begreberne praktiske

Spørg, hvordan »Hvad er et databaseindeks?« gælder i jeres organisation. Et konkret eksempel fra jeres nuværende system er nok til at begynde samtalen.

Se vores ydelser

welcome@engineer.company +45 50 10 61 84 København, Danmark

Tag denne side med dig

Visitkort

Lavet med kærlighed af D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Copyright © 2025 – nu · Engineer ApS · Alle rettigheder forbeholdes.

Udskrevet fra https://engineer.company/da/glossary/database-index/